Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tjaž Brelih

# Preprost prenosni elektrokardiograf (EKG)

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Patricio Bulić

Ljubljana 2015

*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Cilj naloge je izdelati preprost in prenosni elektrokardiograf (EKG), ki je zasnovan na mikrokrmilniku ARM. Pri izdelavi uporabite ceneni modul EKG s tremi elektrodami, elektrokardiogram pa naj se prikazuje na majhnem prikazovalniku OLED.

# Izjava o avtorstvu diplomskega dela

Spodaj podpisani Tjaž Brelih sem avtor diplomskega dela z naslovom:

*Preprost prenosni elektrokardiograf (EKG)*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Patricia Bulića,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 14. septembra 2015 Podpis avtorja:

*Zahvaljujem se družini za vso podporo med študijem, mentorju izr. prof. dr. Patriciu Buliću in asistentu Roku Češnovarju pa za pomoč pri izdelavi diplomskega dela.*

# Kazalo

**Povzetek**

**Abstract**

**1 Uvod** **1**

**2 Elektrokardiografija** **3**

2.1 Elektrokardiogram . . . . . 4

2.2 Uporaba elektrokardiografije . . . . . 6

**3 Strojna oprema** **7**

3.1 Vgrajeni sistemi . . . . . 7

3.2 STM32F4Discovery . . . . . 8

3.3 Modul EKG . . . . . 10

3.4 Modul z zaslonom OLED . . . . . 11

3.5 Povezavanje vseh delov . . . . . 12

**4 Programska oprema** **15**

4.1 Operacijski sistem . . . . . 16

4.2 Inicializacija sistema . . . . . 23

4.3 Zaznavanje QRS kompleksa . . . . . 28

4.4 Izrisovanje na zaslon . . . . . 30

**5 Sklepne ugotovitve** **37**

# Povzetek

Elektrokardiografija je proces spremljanja električne dejavnosti srca. Najpogosteje se uporablja za diagnosticiranje morebitnih obolenj srčne mišice, nudi pa tudi druge možnosti uporabe. Cilj diplomske naloge je prikaz elektrokardiograma in frekvence srčnega utripa na zaslonu. V ta namen smo za spremljanje električne aktivnosti srca uporabili modul EKG, ki s pomočjo treh elektrod pridobi elektrokardiogram srca. Znotraj operacijskega sistema FreeRTOS tečeta dve opravili. Prvo opravilo izvaja preprost algoritem za zaznavo srčnega utripa, ki hkrati meri čase med posameznimi utripi in iz njih izračuna frekvenco srčnega utripa, drugo opravilo pa skrbi za posodabljanje elektrokardiograma in izpis frekvence srčnega utripa na zaslon.

**Ključne besede:** EKG, vgrajeni sistemi, STM32F4Discovery, ARM, FreeRTOS.

# Abstract

Electrocardiography is the process of recording the electrical activity of the heart. It is mostly used for diagnosing a wide variety of possible heart diseases, although there are other possible uses. The aim of this dissertation is to display the electrocardiogram and the heart rate on a screen. For this purpose we utilized an ECG board which uses three electrodes to acquire the electrocardiogram of the heart. There are two tasks running inside the operating system FreeRTOS. The first task contains an algorithm used for detecting the heart beat and measuring the time between the two heart beats. The second task is used to update the data displayed on the screen.

**Keywords:** ECG, embedded systems, STM32F4Discovery, ARM, FreeRTOS.

# Poglavje 1

# Uvod

Z razvojem tehnologije postajajo elektronske naprave zmogljivejše, varčnejše in cenejše, ta trend pa še posebej velja za trg vgrajenih sistemov. Z vse večjo dostopnostjo takih sistemov narašča tudi število produktov, ki take sisteme vsebujejo. Velikokrat želimo predmetom v svoji okolici dodati nekaj pameti. V nekaterih primerih lahko to dosežemo že samo z izpisovanjem določenih informacij na zaslon, v drugih primerih pa vgrajene sisteme uporabljamo za avtomatizacijo procesov. V zadnjem času se pojavlja vse več naprav, ki na tak ali drugačen način skrbijo za naše zdravje. Vse več je naprav, ki jih lahko nosimo na sebi, te pa potem spremljajo množico podatkov o naših telesnih dejavnostih. V preteklosti si je le malokdo predstavljal, da bi lahko v bližnji prihodnosti spremljal elektrokardiogram svojega srca brez obiska ustrezne zdravstvene ustanove.

Osrednji del naše rešitve je razvojna ploščica STM32F4Discovery podjetja STM. Jedro razvojne ploščice tvori procesor Cortex-M4 podjetja ARM, ki preko modula EKG[1] podjetja Sparkfun na zaslon izrisuje elektrokardiogram in frekvenco srčnega utripa. Uporabljali bomo zaslon OLED[2] podjetja Adafruit. Vse te dejavnosti bomo implementirali znotraj operacijskega sistema FreeRTOS, ki bo poskrbel za učinkovito izrabo sistemskih virov in fleksibil-

[1] Elektrokardiograf

[2] Svetleče diode osnovane na organskih spojinah (angl. *organic light-emitting diode*).

nost v primeru nadgradenj strojne ali programske opreme.

V prvem delu diplomskega dela bomo predstavili osnovne informacije o elektrokardiografiji, elektrokardiogramu in možnostih uporabe. V drugem delu bomo predstavili uporabljene strojne komponente, v zadnjem in najobsežnejšem delu te diplomske naloge pa bomo opisali programsko implementacijo.

# Poglavje 2

# Elektrokardiografija

Elektrokardiografija je proces spremljanja električne dejavnosti srca z uporabo elektrod, ki se pritrdijo na telo. Te elektrode zaznavajo spremembe električnega potenciala na koži zaradi depolarizacije srčne mišice. Najpogosteje srečamo konfiguracijo z 10 elektrodami, kjer 4 od teh elektrod pritrdimo na vsako izmed okončin, preostalih 6 pa na prsni koš. Ena izmed elektrod mora biti vedno prisotna kot ozemljitev. Iz teh 10 elektrod nato tvorimo 12 *odvodov*. Izraz *odvod* se v elektrokardiografiji uporablja za vektorje, vzdolž katerih merimo depolarizacijo srčne mišice. Vsak odvod predstavlja razliko električnih potencialov med dvema različnima točkama na telesu. Te točke so največkrat kar posamezne elektrode, v nekaterih primerih pa so točke kombinacija večih elektrod. V konfiguraciji z 12 odvodi tako merimo 12 različnih vektorjev, ki jih pridobimo s pomočjo prej omenjenih elektrod [1].

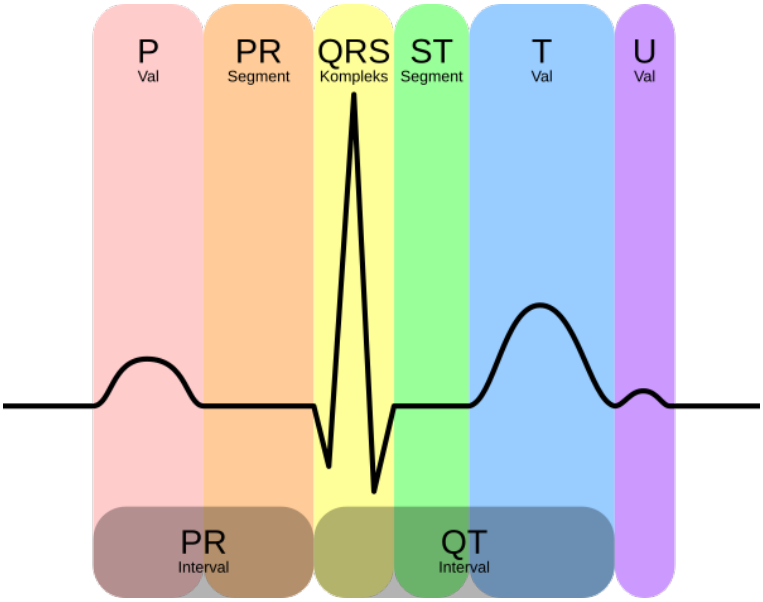
Čeprav je konfiguracija z 10 elektrodami najpogostejša, pa še zdaleč ni edina. Manj elektrod oziroma manj odvodov pomeni tudi manjšo natančnost meritev, vendar lahko s posebnimi načini obdelave podatkov iz odvodov to izgubo močno zmanjšamo [2][3]. V primeru manjšega števila elektrod se lahko zgodi, da nekaterih lastnosti posameznih odsekov srčnega impulza enostavno ne moremo zaznati. Tako na primer brez prekordialnih odvodov (odvodi, ki jih dobimo s pomočjo elektrod na prsnem košu) ne moremo zaznati relativne višine ST segmenta [4]. Poznamo tudi konfiguracije z več kot 10 elektrodami

oziroma 12 odvodi, vendar so te konfiguracije uporabne samo za namene diagnosticiranja specifičnih težav srčne mišice, primarno za lažjo prepoznavo miokardnega infarkta, bolje znanega kot srčni napad [5][6].

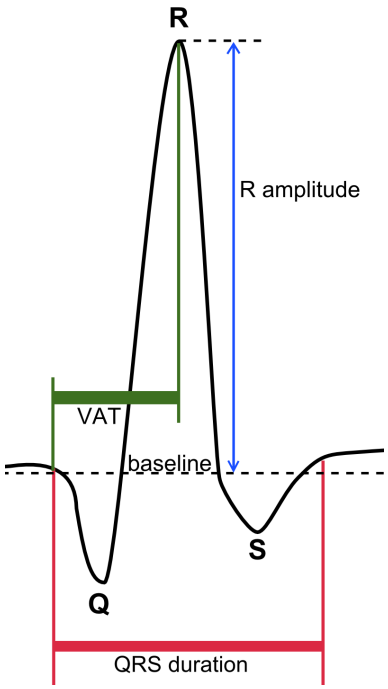
## 2.1 Elektrokardiogram

Graf električne napetosti v odvisnosti od časa imenujemo elektrokardiogram. Z enim elektrokardiogramom lahko ponazorimo en odvod. V primeru konfiguracije z 12 odvodi imamo tako 12 elektrokardiogramov, vsak pa prikazuje malenkost drugačen graf, saj vsak odvod meri električno napetost srčne mišice z drugačnega zornega kota. Tako dobimo vpogled v delovanje posameznih anatomskih delov srčne mišice, kar nam omogoča lažjo diagnozo potencialnih obolenj.

Elektrokardiogram razdelimo na več odsekov (slika 2.1), vsak odsek pa predstavlja depolarizacijo ali polarizacijo določenih delov srčne mišice. P val odraža depolarizacijo preddvorov, QRS kompleks odraža depolarizacijo prekatov, T val pa odraža polarizacijo prekatov. QRS kompleks lahko nadalje razdelimo na Q, R in S valove (slika 2.2). Bolezenska stanja srčne mišice največkrat diagnosticiramo s preučevanjem amplitude, trajanja ali zamikov posameznih odsekov elektrokardiograma relativno glede na preostale odseke. Pomembna metrika je tudi R-R interval, ki meri čas med posameznimi R valovi. Frekvenca srčnega utripa je izpeljana iz R-R intervala, vendar se v komercialnih izdelkih navadno prilagaja glede na prejšnje vrednosti. Na ta način dosežemo manjša nihanja frekvence srčnega utripa na račun manjše natančnosti. Raziskave so pokazale, da lahko iz podatkov o R-R intervalih zaradi večje natančnosti izvemo več kot samo iz frekvence srčnega utripa [7].




Slika 2.1: Prikaz običajnega elektrokardiograma. Povzeto po [1].




Slika 2.2: Shematski prikaz QRS kompleksa [8].

## 2.2 Uporaba elektrokardiografije

Elektrokardiografijo najpogosteje uporabljamo za diagnosticiranje obolenj srčne mišice. Eden izmed primerov take uporabe je odkrivanje nemih srčnih bolezni pri mladih športnikih [9].

Uporabljamo pa jo lahko tudi v druge namene. Obstaja povezava med dihanjem in dolžino R-R intervalov, imenovana respiratorna sinusna aritmija. Pri njej se R-R interval med vdihovanjem skrajša, med izdihovanjem pa podaljša [10]. Elektrokardiografija se lahko uporablja za zaznavanje obstruktivne spalne apneje, kjer bolnik med spanjem zaradi zapore v zgornjih dihalnih poteh pogostokrat preneha dihati [11].

Znanstveniki so uspešno prikazali način uporabe elektrokardiografije za namene identificiranja posameznikov [12].

# Poglavje 3

# Strojna oprema

Diplomska naloga temelji na zmogljivi in cenovno ugodni razvojni ploščici STM32F4Discovery, ki podatke pridobiva preko modula EKG, nato pa jih izrisuje na zaslon OLED.

## 3.1 Vgrajeni sistemi

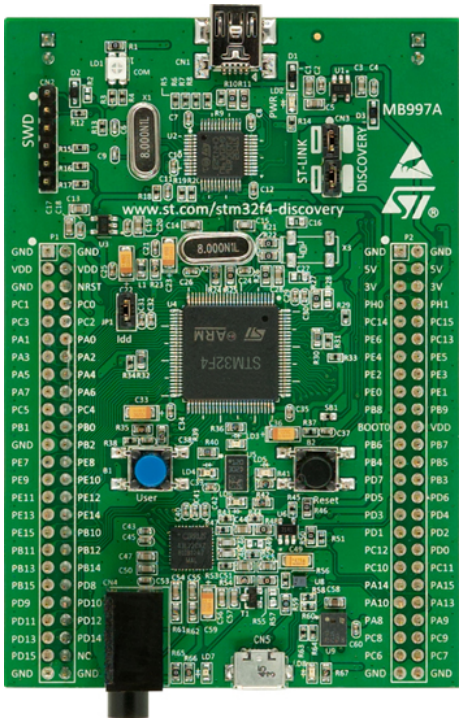
Vgrajen sistem je računalniški sistem, ki v nasprotju s splošno namenskim računalniškim sistemom, kot je recimo prenosni računalnik, opravlja točno določeno nalogo in je pogostokrat vgrajen v neko drugo napravo. Ker tak sistem opravlja točno določeno nalogo, lahko delovanje le-tega zelo optimiziramo in na ta način zmanjšamo porabo električne energije, znižamo ceno in zmanjšamo fizično velikost.

Število prodanih vgrajenih sistemov je veliko večje v primerjavi s številom prodanih klasičnih računalniških sistemov. Svojo okolico želimo ustvariti čim pametnejšo, zato v vsakdanje predmete vedno pogosteje vgrajujemo majhne računalnike. Velikokrat je razlog za vgradnjo avtomatizacija določenih procesov. Tako lahko na primer izdelamo sistem za samodejno zalivanje rož, ki za razliko od človeka ni pozabljiv, poleg tega pa se tak sistem lahko prilagaja na določene parametre, kot je na primer vlažnost prsti.

Pomemben koncept v svetu vgrajenih sistemov so tudi prekinitve. Preki-

nitve so signal, ki ga procesorju pošlje strojna ali programska oprema. Z njimi procesorju sporočimo, da se je zgodil dogodek, ki potrebuje njegovo takojšnje posredovanje. Procesor nato shrani svoje trenutno stanje in prične z izvajanjem prekinitveno servisnega programa. Prekinitveno servisni program je običajno kratek in pogosto je trajanje njegovega izvajanja časovno omejeno. Prekinitvam, ki jih sproži programska koda, pravimo tudi pasti [13].

## 3.2 STM32F4Discovery




Slika 3.1: Razvojna ploščica STM32F4Discovery [14].

Osnova razvojne ploščice STM32F4Discovery (slika 3.1) je mikrokontroler *STM32F407VGT6*. Poleg zmogljivega mikrokontrolerja razvojna ploščica vsebuje še merilnik pospeška, digitalni mikrofon, digitalno-analogni avdio pretvornik s 3.5mm avdio priključkom, 4 uporabniške svetleče diode (angl.

*LED - light-emitting diode*), 1 uporabniški gumb, USB mikro-AB priključek in 82 splošno namenskih vhodno-izhodnih priključkov. Širok nabor funkcionalnosti in nizka cena razvojne ploščice omogočata hiter in učinkovit razvoj prototipov [14].

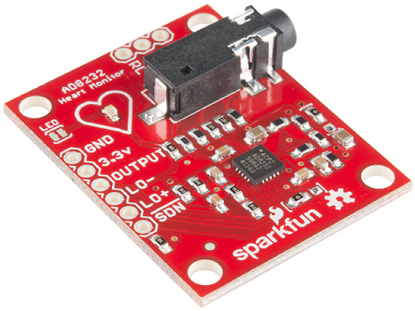
Jedro mikrokontrolerja tvori visoko zmogljivi procesor ARM Cortex-M4, ki izvaja ukaze s frekvenco do 168 MHz. Namenjen je predvsem nadzoru in obdelavi digitalnih signalov [15]. Mikrokontroler poleg procesorskega jedra vsebuje še 192K delovnega pomnilnika SRAM, 1MB trajnega pomnilnika Flash ter širok nabor perifernih in komunikacijskih enot:

- CRC enoto
- dva DMA krmilnika
- tri 12-bitne ADC pretvornike
- dva 12-bitna DAC pretvornika
- DCMI[1] vmesnik
- štirinajst časovnikov
- kriptografski procesor
- generator naključnih števil
- zgoščevalni procesor
- uro realnega časa
- tri $I^2C$ komunikacijske vmesnike
- tri SPI komunikacijske vmesnike
- dva $I^2S$ komunikacijska vmesnika
- štiri USART komunikacijske vmesnike

[1]Vmesnik za digitalno kamero (angl. *digital camera interface*).

- dva UART komunikacijska vmesnika
- SDIO[2] komunikacijski vmesnik
- dva CAN[3] komunikacijska vmesnika
- Ethernet MAC 10/100 komunikacijski vmesnik z vgrajenim DMA krmilnikom
- USB OTG vmesnik
- FSMC[4] krmilnik

## 3.3 Modul EKG




Slika 3.2: Modul EKG podjetja Sparkfun [16].

V naši diplomski nalogi smo uporabili modul EKG, osrednji del modula pa je čip *AD8232* [16] (slika 3.2). Modul proizvaja podjetje Sparkfun, čip *AD8232* pa podjetje Analog Devices. Na modulu se nahaja 3.5 mm priključek, na

[2]Vmesnik za spominske kartice (angl. *secure digital input/output interface*).

[3]Vmesnik za komunikacijo med mikrokontrolerji v vozilih (angl. *controller area network*).

[4]Krmilnik za naprave s statičnim pomnilnikom (angl. *flexible static memory controller*).

katerega priklopimo elektrode. Čip podpira samo konfiguracijo s tremi elektrodami, kjer se ena elektroda uporablja za ozemljitev, s pomočjo preostalih dveh pa tvorimo en odvod.

Modul ni certificirana medicinska naprava, zato ni namenjen diagnosticiranju in zdravljenju morebitnih zdravstvenih težav.

Na modulu se nahaja devet nožic. Dve nožici se uporabljata za napajanje modula, tri nožice so namenjene spremljanju signalov elektrod, preostale štiri pa so namenjene podatkovni povezavi modula in razvojne ploščice. Modul preko nožice `Output` prenaša odvod kot spremembo električne napetosti, preko nožic `LO-` in `LO+` lahko zaznamo, kdaj so elektrode pritrjene na telo, nožica `SDN` pa nam daje možnost, da modul ugasnemo. Na modulu se nahaja tudi svetleča dioda, ki odraža električno napetost na nožici `Output`.

## 3.4 Modul z zaslonom OLED


Slika 3.3: Zaslon OLED podjetja Adafruit. Povzeto po [17].

Za prikaz smo uporabili enobarvni zaslon OLED z diagonalo 3.3 cm podjetja Adafruit (slika 3.3). Zaslon tvori 128x64 posameznih pikslov bele barve, ki v nasprotju z običajnimi zasloni LED ne potrebujejo dodatne osvetlitve. To pomeni zmanjšano porabo ter izboljšan kontrast.

Z zaslonom upravlja čip *SSD1306* [17], s katerim lahko komuniciramo

preko protokola SPI[5] ali $I^2C$[6]. V našem primeru bomo uporabljali protokol SPI. Zaslon za delovanje potrebuje napetost 3.3V, vendar ga lahko priključimo tudi na napetost 5V, saj ima modul vgrajen 3.3V regulator, ki pretvori vse logične nivoje na ustrezno napetost.

Poleg treh nožic za napajanje ima modul še pet podatkovnih nožic. Nožici `Data` in `Clk` se uporabljata za prenos podatkov med razvojno ploščico in kontrolnim čipom, kjer se preko nožice `Data` pošiljajo podatki bit po bit, preko nožice `Clk` pa se pošilja urin signal. Nožica `DC` določa, ali naj se poslani podatki obravnavajo kot ukaz ali kot podatki. Preko nožice `Rst` lahko čip ponastavimo na privzete vrednosti. Zaradi načina delovanja protokola SPI, kjer je lahko na vodili `Data` in `Clk` povezanih več naprav, moramo čipu sporočiti, kdaj so podatki namenjeni njemu. To storimo preko nožice `CS`.

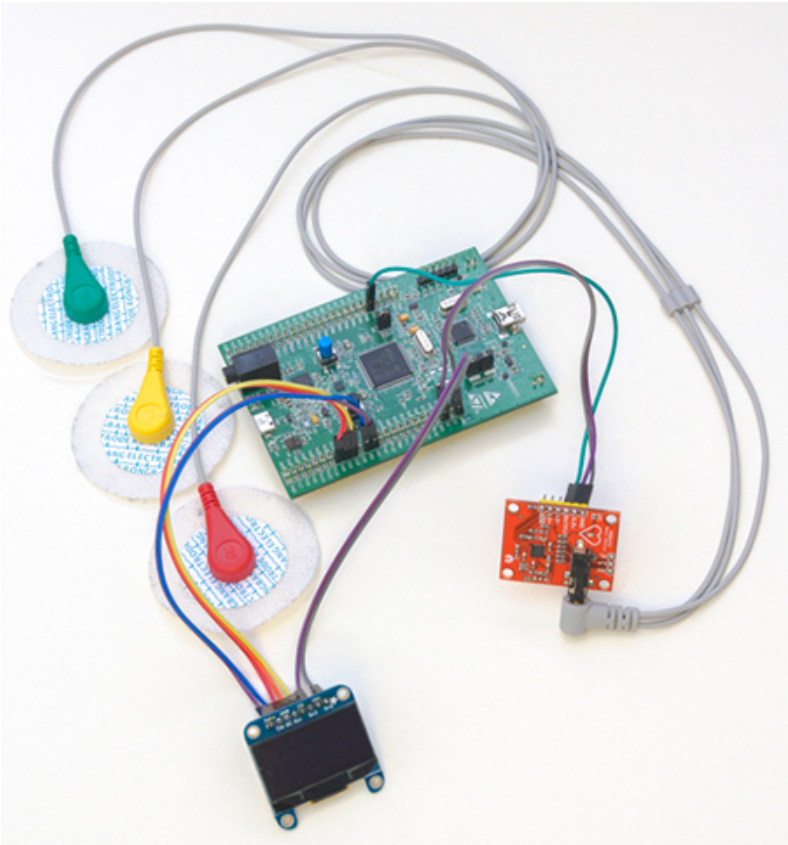
## 3.5 Povezavanje vseh delov

Slika 3.4 prikazuje celoten sistem z obema moduloma povezanima na razvojno ploščico in tremi elektrodami, ki se pričvrstijo na telo.

Najprej oba modula povežemo na priključka za ozemljitev in napetost. Nato nožico `Output` modula EKG povežemo na razvojno ploščico na priključek `PC2`. Nožici `Data` in `Clk` modula z zaslonom povežemo na priključka `PB5` in `PB3`, nožice `DC`, `Rst` in `CS` pa povežemo na priključke `PD0`, `PD2` in `PD4`.

[5]Serijski periferni vmesnik (angl. *serial peripheral interface)*

[6]Protokol za komunikacijo med integriranimi vezji (angl. *inter-integrated circuit)*



Slika 3.4: Slika sistema.

# Poglavje 4

# Programska oprema

Programsko kodo, ki teče na razvojni ploščici, smo razvili s pomočjo razvojnega okolja *IAR Embedded Workbench* [18]. Podjetje IAR ponuja dve brezplačni licenci, kjer je prva licenca časovno omejena na 30 dni uporabe, druga pa je omejena z velikostjo izhodnega programa na 32 KB. Poleg časovne ali velikostne omejitve obema brezplačnima licencama manjka še nekaj drugih funkcionalnosti, ki jih pa ne bomo potrebovali. V svetu vgrajenih sistemov je 32 KB relativno veliko prostora, zato je za naše potrebe licenca z omejitvijo velikosti izhodnega programa povsem zadovoljiva. Programsko kodo smo spisali v programskem jeziku *C*.

Enega izmed časovnikov nastavimo tako, da vsakih nekaj milisekund sproži dogodek. Ta dogodek sporoči analogno-digitalnemu pretvorniku, naj preko priključka `PC2` prebere vrednost nožice `Output` na modulu EKG. Ko analogno-digitalni pretvornik to vrednost iz analogne oblike pretvori v digitalni zapis, pošlje krmilniku za neposreden dostop do pomnilnika sporočilo, da naj to vrednost prenese na določeno mesto v pomnilniku. Ves ta proces se zgodi brez sodelovanja procesorja, kar zmanjša njegovo obremenjenost. Ta vrednost se nato izriše na zaslon, hkrati pa se uporabi za zaznavanje QRS kompleksa. Čas med dvema QRS kompleksoma merimo s časovnikom, ki vsako milisekundo sproži prekinitev, prekinitveno servisni program pa nato spremenljivko `heartBeatMS` poveča za ena. Ko sistem zazna QRS kompleks,

se trenutna vrednost omenjene spremenljivke najprej prepiše v lokalno spremenljivko, zatem pa ponastavi na vrednost nič.

## 4.1 Operacijski sistem

Operacijski sistem je sistemski program, ki upravlja s strojno in programsko opremo računalnika ter nudi storitve računalniškim programom. Osnovna funkcija operacijskega sistema je prekraplanje med različnimi programi. Če operacijski sistem med temi programi preklaplja dovolj hitro, se uporabniku zdi, kot da se programi izvajajo istočasno. Na ta način lahko na primer hkrati brskamo po spletu in poslušamo glasbo. Pred prihodom operacijskih sistemov se je ob določenem času lahko na računalniku izvajal samo en program. Operacijski sistem uporabniku omogoča tudi kontrolo nad izvajanjem programov, saj lahko uporabnik v kateremkoli trenutku zažene katerikoli program. Zaradi zasnove splošno namenskih računalnikov lahko na njem izvajamo raznorazne programe, potrebujemo pa le datoteke, ki ta program opisujejo. Tako lahko uporabnik v nekem trenutku zažene program za brskanje po spletu, v naslednjem trenutku program za urejanje besedil, zatem odpre program za spremljanje elektronske pošte, potem pa lahko uporabnik zapre vse do sedaj odprte programe in odpre program za predvajanje videoposnetkov.

Dandanes imajo praktično vsi računalniki nameščen operacijski sistem, uporabljajo pa ga tudi nekatere novejše naprave, kot so mobilni telefoni, igralne konzole, televizorji in podobni.

Operacijske sisteme lahko uporabimo tudi v vgrajenih sistemih, predvsem ko razvijamo kompleksnejše rešitve. Jedro operacijskih sistemov za vgrajene sisteme se ne razlikuje močno od jeder bolj znanih operacijskih sistemov, ki jih najdemo na splošno namenskih računalnikih. Njegova osnovna funkcija preklapljanja med programi ostaja enaka. Operacijski sistemi za splošno namenske računalnike poleg preklapljanja med programi dandanes podpirajo tudi ogromno množico drugih funkcionalnosti, ki pa jih v svetu vgrajenih

sistemov ne potrebujemo. Poleg tega morajo moderni operacijski sistemi delovati tudi na široki paleti različnih računalniških specifikacij in podpirati ogromno količino različnih naprav, zato so temu primerno kompleksnejši, skupaj s kompleksnostjo pa naraščajo tudi sistemske zahteve.

### 4.1.1 FreeRTOS

V diplomski nalogi smo uporabili brezplačni realno-časovni operacijski sistem FreeRTOS. FreeRTOS je operacijski sistem za vgrajene sisteme, ki je kljub svoji zmogljivosti prostorsko zelo učinkovit. Jedro operacijskega sistema se nahaja v samo treh datotekah, skoraj v celoti pa je spisano v programskem jeziku *C*.

#### Opravilo

V FreeRTOS žargonu programom pravimo opravila. Opravila so funkcije, ki se nikoli ne smejo zaključiti, kar zagotovimo z uporabo neskončnih zank.

Opravilo se lahko nahaja v enem izmed štirih stanj. Ko je opravilo v stanju *Running* se njegova koda izvaja na procesorju. V vsakem trenutku je lahko v tem stanju samo eno opravilo.

Koda opravila lahko kliče neko funkcijo, kjer nato čakamo na nek dogodek. Taki funkciji pravimo blokirajoča funkcija. Dokler se omenjeni dogodek ne zgodi, je opravilo v stanju *Blocked*. Blokirajoče funkcije lahko uporabljamo za medsebojno sinhronizacijo večih opravil. Opravilo lahko za določen čas blokiramo s funkcijo `vTaskDelay`.

Če znotraj opravila kličemo funkcijo `vTaskSuspend`, se opravilo postavi v stanje *Suspended*. Dokler je opravilo v stanju *Suspended* se njegova koda ne bo izvajala, iz tega stanja pa lahko opravilo spravimo samo s klicem funkcije `vTaskResume`.

V kolikor se opravilo trenutno ne izvaja, hkrati pa ni ne v stanjih *Blocked* ali *Suspended*, je v stanju *Ready*. Opravilo tedaj čaka, da pride na vrsto za izvajanje, saj se, kot rečeno, lahko hkrati izvaja samo eno opravilo.

Opravilo ustvarimo s klicem funkcije `xTaskCreate`. Opravilo se ne sme nikoli zaključiti, lahko pa ga uničimo s klicem funkcije `vTaskDelete`.

V praksi za komunikacijo z določeno periferno napravo uporabljamo samo eno opravilo (angl. *gatekeeper task*). Vsa ostala opravila, ki hočejo dostopati do te periferne naprave, to storijo preko opravila, ki je zadolženo za omenjeno periferno napravo.

V naši diplomski nalogi smo uporabili dve opravili. Prvo opravilo skrbi za izris na zaslon, drugo opravilo pa je namenjeno zaznavanju QRS kompleksa in izračunu frekvence srčnega utripa. Obe opravili imata enako prioriteto. Ustvarjanje obeh opravil je prikazano na izseku kode 4.1.

Izsek kode 4.1: Ustvarjanje obeh opravil.

```
xTaskCreate(
    vTaskZaslonMain,                // Kazalec na funkcijo opravila
    (const signed char*) "Task1",   // Ime opravila
    configMINIMAL_STACK_SIZE,       // Velikost sklada
    (void*) NULL,                   // Kazalec na argumente funkcije
    tskIDLE_PRIORITY + 2UL,         // Prioriteta opravila
    NULL                            // Oprimek opravila
);

xTaskCreate(
    vTaskHeartBeat,                 // Kazalec na funkcijo opravila
    (const signed char*) "Task2",   // Ime opravila
    configMINIMAL_STACK_SIZE,       // Velikost sklada
    (void*) NULL,                   // Kazalec na argumente funkcije
    tskIDLE_PRIORITY + 2UL,         // Prioriteta opravila
    NULL                            // Oprimek opravila
);
```

Osnovna funkcija operacijskega sistema je tudi podpora komunikaciji med različnimi programi. V FreeRTOS imamo tako na voljo vrste, binarne semaforje, številne semaforje, ključavnice in rekurzivne ključavnice. Omenjene podatkovne strukture lahko v nekaterih primerih uporabimo tudi za sinhronizacijo različnih opravil.

### Vrsta

Vrsta je podatkovna struktura, kjer se podatki obravnavajo po principu prvi pride, prvi melje (angl. *FIFO - first in, first out*). Prvi podatek, ki smo ga v vrsto vstavili, se bo iz vrste tudi prvi prebral. Vrsta ima končno velikost, kar pomeni, da lahko vanjo vstavimo samo končno število elementov. Ker imamo v vgrajenih sistemih mnogokrat na voljo malo delovnega pomnilnika, moramo velikost vrste izbrati skrbno.

Vrsto lahko uporabljajo vsa opravila. V praksi najpogosteje srečamo situacijo, kjer eno opravilo v vrsto piše, drugo pa iz nje bere. Velikokrat srečamo tudi situacijo, kjer eno opravilo iz vrste bere, več opravil pa vanjo piše. Tako konfiguracijo največkrat uporabljamo z opravili, ki želijo dostopati do periferije. Na ta način lahko na primer z zaslonom upravlja samo eno opravilo, ostala opravila, ki želijo na zaslon izpisati podatke, pa preko vrste te podatke dostavijo opravilu, ki upravlja z zaslonom.

Vrsto ustvarimo s klicem funkcije `xQueueCreate`, izbrišemo pa jo lahko s klicem funkcije `vQueueDelete`. Podatke v vrsto vstavljamo z uporabo funkcije `xQueueSendToBack`, iz vrste pa podatke preberemo s klicem funkcije `xQueueRecieve`.

V kolikor opravilo želi brati iz vrste, ki je prazna, ali pa pisati v vrsto, ki je polna, se opravilo postavi v stanje *Blocked*. Opravilo ostane v tem stanju dokler niso izpolnjeni ustrezni pogoji, ali pa je pretekel ustrezen čas.

V naši diplomski nalogi smo uporabili eno vrsto, ki lahko hkrati hrani dva podatka velikosti `unsigned char*`. S pomočjo te vrste se med opraviloma prenaša podatek o frekvenci srčnega utripa. V vrsto piše opravilo, ki izračuna frekvenco srčnega utripa, drugo opravilo pa ta podatek iz vrste prebere in ga prikaže na zaslonu. Ustvarjanje vrste je prikazano na izseku kode 4.2.

Spremenljivka tipa `xQueueHandle` je globalna spremenljivka, kar pomeni, da lahko do nje dostopamo iz katerekoli lokacije v programu.

Izsek kode 4.2: Ustvarjanje vrste.

```
1 xQueueHandle queueHeartBeat = 0;
2
3 queueHeartBeat = xQueueCreate( 2, sizeof( unsigned char* ) );
4 if( queueHeartBeat == 0 ) {
5     // Prišlo je do napake pri ustvarjanju vrste
6     while( 1 );
7 }
```

### Semaforji

V FreeRTOS poznamo binarne in številne semaforje. Binarni semafor je vrsta, ki hrani en sam podatek. Ta podatek je navadno preprostega tipa, njegova vrednost pa nas ne zanima. V najpreprostejšem primeru eno opravilo v semafor piše, drugo pa iz njega bere. Binarne semaforje lahko uporabimo za sinhronizacijo med opravili ali pa za implementacijo odloženih prekinitev (angl. *deferred interrupt*). Odložene prekinitve so reakcije na prekinitev, kjer prekinitveno servisni program to nalogo preloži opravilu. To lahko storimo tako, da prekinitveno servisni program v semafor zapiše nek podatek, opravilo pa kasneje ta podatek iz semaforja prebere. V FreeRTOS žargonu ob pisanju vrednosti v semafor pravimo, da smo semafor vzeli, ko pa podatek iz semaforja preberemo pa pravimo, da smo semafor dali nazaj. S prisotnostjo podatka v semaforju opravilo ve, da je prišlo do prekinitve, zato lahko začne s servisiranjem prekinitve. Ker morajo biti prekinitveno servisni programi čim krajši, lahko s pomočjo binarnih semaforjev tako uporabljamo tudi daljše in kompleksnejše servisne programe, ne da bi motili izvajanje preostalih opravil.

Številni semafor je vrsta, ki lahko v nasprotju z binarnim semaforjem hrani več podatkov hkrati. Tako kot pri binarnih semaforjih nas tudi pri številnih semaforjih vrednosti podatkov v semaforju ne zanimajo. Uporabljamo jih predvsem takrat, ko nas zanima, koliko prekinitev se je zgodilo v času, ko se opravilo ni izvajalo. Ob vsaki prekinitvi se v številni semafor vpiše nov podatek, nato pa opravilo prešteje vse nove podatke v semaforju.

Binarni semafor ustvarimo s funkcijo `xSemaphoreCreateBinary`, uničimo pa ga lahko s funkcijo `vSemaphoreDelete`. Podatek v semafor zapišemo s

funkcijo `xSemaphoreTake`, beremo pa z uporabo funkcije `xSemaphoreGive`.

Števni semafor ustvarimo s klicem funkcije `xSemaphoreCreateCounting`, uničimo pa ga z isto funkcijo kot binarni semafor. Za pisanje in branje podatkov iz števnega semaforja uporabljamo iste funkcije kot za uporabo binarnega semaforja.

### Ključavnice

FreeRTOS nam omogoča uporabo navadnih in rekurzivnih ključavnic (angl. *mutex - mutual exclusion*). Navadne ključavnice so binarni semaforji, ki pa jih uporabljamo za drugačne namene. Uporabljamo jih predvsem za nadzor dostopa do skupnih virov. Opravilo pred uporabo skupnega vira zaklene ključavnico in na ta način onemogoči ostalim opravilom, da bi dostopala do istega vira. Ko opravilo preneha z uporabo tega vira, ključavnico spet odklene. Na ta način se izognemo morebitnim nepravilnostim, do katerih bi lahko prišlo ob skupnem dostopu večih opravil do istega vira. Za uporabo navadnih ključavnic uporabljamo iste funkcije kot za uporabo semaforjev [19].

Rekurzivne ključavnice so števni semaforji, ki pa jih uporabljamo za drugačne namene. Navadne ključavnice lahko zaklenemo samo enkrat, rekurzivne pa večkrat. Rekurzivno ključavnico moramo odkleniti točno tolikokrat, kolikorkrat smo jo zaklenili.

Ključavnico ustvarimo s funkcijo `xSemaphoreCreateMutex`, uničimo pa jo lahko s klicem funkcije `xSemaphoreDelete`. Za zaklepanje in odklepanje ključavnice uporabimo isti funkciji kot za uporabo binarnih semaforjev.

Z uporabo funkcije `xSemaphoreCreateRecursiveMutex` ustvarimo novo rekurzivno ključavnico, uničimo pa jo na enak način kot navadno ključavnico. Rekurzivno ključavnico zaklenemo s funkcijo `xSemaphoreTakeRecursive`, odklenemo pa jo lahko s klicem funkcije `xSemaphoreGiveRecursive`.

### Preklapljanje opravil

Preklapljanje opravil je osnovni del jedra operacijskega sistema. Na ta način dosežemo iluzijo sočasnega izvajanja večih programov. Del strojne opreme

periodično proži prekinitve z najvišjo prioriteto (angl. *systick - system tick*), kjer v prekinitveno servisnem programu nato zamenjamo izvajajoče opravilo. Periodično proženje prekinitev dosežemo z uporabo sistemskega časovnika. Časovnik vsako urino periodo vrednost v svojem registru poveča za ena, vse dokler se vrednost v tem registru ne ujema z vrednostjo v primerjalnem registru. Takrat se sproži prekinitev, hkrati pa se notranji register časovnika ponastavi na vrednost 0. Vrednost v primerjalnem registru lahko spreminjamo, posledično pa spreminjamo tudi količino časa, ki preteče med dvema prekinitvama. Tak sistemski časovnik je sestavni del vseh jeder tipa Cortex, vsa jedra pa si delijo tudi prekinitveni krmilnik. Posledica tega je, da je programska koda med vsemi jedri tipa Cortex prenosljiva, kar pomeni, da lahko programsko kodo spišemo za jedro Cortex-M2, ta koda pa brez sprememb deluje na vseh ostalih jedrih tipa Cortex.

Zgoraj omenjeni način preklapljanja opravil pa ima eno pomanjkljivost. V kolikor prekinitev sistemskega časovnika prekine izvajanje nekega drugega prekinitveno servisnega programa, se ob koncu preklapljanja opravila ne bomo vrnili v prekinjen prekinitveno servisni program, ampak bo procesor začel izvajati opravilo, ki je na vrsti za izvajanje. Rešitev za to težavo je uporaba sistemske prekinitve *pendSV* (angl. *pended system call*). Ko sistemski časovnik sproži prekinitev, v njenem prekinitveno servisnem programu pokličemo funkcijo, ki sproži prekinitev *pendSV*. Ker ima prekinitev *pendSV* najnižjo možno prioriteto, ta ne bo nikoli prekinila drugega prekinitveno servisnega programa. Zdaj lahko varno preklopimo med opravili, saj vemo, da nismo prekinili nobenega prekinitveno servisnega programa. Še vedno pa se lahko zgodi, da se med preklapljanjem opravila sproži kakšna druga prekinitev z višjo prioriteto. To težavo odpravimo tako, da na začetku izvajanja prekinitveno servisnega programa za prekinitev *pendSV* onemogočimo vse prekinitve. Seveda moramo prekinitve ob zaključku prekinitveno servisnega programa spet omogočiti.

### Razvrščevalnik

Naloga razvrščevalnika je, da izbere opravilo, ki se bo izvajalo naslednje. Razvrščevalnik določi tudi, koliko časa se bo posamezno opravilo izvajalo, preden bo na vrsti naslednje opravilo. V najpreprostejšem primeru se opravila izvajajo ciklično eno za drugim, vsakemu opravilu pa je dodeljena enaka količina časa. Na tem principu deluje tudi razvrščevalnik v FreeRTOS.

Vsa opravila v FreeRTOS imajo določeno prioriteto. Razvrščevalnik bo za izvajanje vedno izbral opravilo z najvišjo prioriteto, ki je v stanju *Ready*. Opravila v stanjih *Blocked* ali *Suspended* ne pridejo na vrsto za izvajanje, dokler ne pridejo v stanje *Ready*. V kolikor razvrščevalnik naleti na situacijo, kjer nobeno opravilo ni v stanju *Ready*, za izvajanje določi opravilo *IDLE*. Opravilo *IDLE* je sestavljeno samo iz ene prazne neskončne zanke. V kolikor želimo, da opravilo *IDLE* počne kaj uporabnega, lahko definiramo funkcijo `vApplicationIdelHook` in znotraj nje napišemo kodo, ki se potem izvaja v opravilu *IDLE*. Znotraj te funkcije pa ne smemo klicati funkcij, ki bi lahko opravilo *IDLE* blokirale, ali klicati funkcije `vTaskSuspend`, ki opravilo suspendira. Programska koda v funkciji `vApplicationIdelHook` lahko celoten sistem postavi v stanje nizke porabe. Na ta način lahko dodatno zmanjšamo porabo električne energije našega sistema.

Opravilo lahko svoj dodeljeni čas tudi prepusti naslednjemu opravilu, še preden ta poteče. To storimo s klicem funkcije `taskYIELD`.

## 4.2 Inicializacija sistema

Preden lahko začnemo z izvajanjem opravil, moramo nastaviti parametre naprav, ki jih bomo med izvajanjem uporabljali. Tako zagotovimo ustrezno delovanje vseh naprav. Najprej moramo vsaki napravi, ki jo bomo potrebovali, omogočiti urin signal. Na ta način naprave, ki jih ne potrebujemo, ostanejo neaktivne, kar zmanjša porabo električne energije.

### 4.2.1 Časovnik

Potrebovali bomo dva časovnika. Prvi časovnik bomo nastavili tako, da bo na vsakih nekaj milisekund sprožil prenos podatka iz naprave *ADC3* preko krmilnika *DMA2* v delovni pomnilnik. Drugi časovnik bo štel število preteklih milisekund med dvema zaznanima QRS kompleksoma.

Izsek kode 4.3 prikazuje programsko kodo, ki nastavi parametre prvega časovnika. Najprej z ukazom `RCC_APB1PeriphClockCmd` omogočimo vhodno uro časovnika. Vhodna ura v časovnik teče s frekvenco 168 MHz. To frekvenco nato delimo s 4, kar pomeni, da naš časovnik dejansko deluje pri frekvenci 42 MHz. Programski delilnik frekvence nato to frekvenco deli z 42000, kar pomeni, da se notranji register časovnika poveča za ena vsako milisekundo. Ko ta register doseže vrednost `EKG_REFRESH_RATE_MS`, se sproži dogodek `TIM_TRGOSource_Update`. Analogno-digitalni pretvornik bomo nato nastavili tako, da bo ob dogodku `TIM3_TRGO` začel s pretvorbo iz analognega v digitalni zapis.

Izsek kode 4.3: Inicializacija prvega časovnika.

```
1  RCC_APB1PeriphClockCmd( RCC_APB1Periph_TIM3, ENABLE );
2
3  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
4
5  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV4;
6  TIM_TimeBaseStructure.TIM_Prescaler = 42000 - 1;
7  TIM_TimeBaseStructure.TIM_Period = EKG_REFRESH_RATE_MS;
8  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
9
10 TIM_TimeBaseInit( TIM3, &TIM_TimeBaseStructure );
11
12 TIM_SelectOutputTrigger( TIM3, TIM_TRGOSource_Update );
13
14 TIM_Cmd( TIM3, ENABLE );
```

Izsek kode 4.4 prikazuje nastavitve drugega časovnika. Tako kot prvi časovnik tudi ta poveča svoj notranji register vsako milisekundo. Za razliko od prvega časovnika pa drugi časovnik šteje samo do ena, nato pa sproži prekinitev. V prekinitveno servisnem programu (izsek kode 4.5) najprej preverimo, ali je do prekinitve dejansko prišlo, nato pa spremenljivko `heartBeatMS`

povečamo za ena. Na koncu še odstranimo zahtevo za prekinitev, saj bi se v nasprotnem primeru prekinitveno servisni program izvajal v nedogled.

Izsek kode 4.4: Inicializacija drugega časovnika.

```
RCC_APB1PeriphClockCmd( RCC_APB1Periph_TIM4, ENABLE );

TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV4;
TIM_TimeBaseStructure.TIM_Prescaler = 42000 - 1;
TIM_TimeBaseStructure.TIM_Period = 1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

TIM_TimeBaseInit( TIM4, &TIM_TimeBaseStructure );

TIM_ITConfig( TIM4, TIM_IT_Update, ENABLE );

NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

NVIC_Init( &NVIC_InitStructure );

TIM_Cmd( TIM4, ENABLE );
```

Izsek kode 4.5: Prekinitveno servisni program za štetje milisekund.

```
extern uint16_t heartBeatMS;

void TIM4_IRQHandler() {
    if( TIM_GetITStatus( TIM4, TIM_IT_Update ) ) {
        heartBeatMS++;
    }

    TIM_ClearITPendingBit( TIM4, TIM_IT_Update );
}
```

### 4.2.2 Analogno-digitalni pretvornik

Analogno-digitalni pretvornik potrebujemo za pretvorbo električne napetosti v digitalno vrednost. V našem primeru električna napetost na vhodu v analogno-digitalni pretvornik niha med 0 in 3.3 volti. Ko je napetost 0 voltov,

je tudi pretvorjena digitalna vrednost enaka 0. Ko pa napetost naraste na 3.3 volte, se digitalna vrednost nastavi na vrednost, ki je odvisna od nastavljene resolucije. Resolucijo lahko nastavimo na 6, 8, 10 ali 12 bitov. V primeru, da uporabljamo resolucijo velikosti 6 bitov, se napetost 3.3 volte pretvori v vrednost 63, ko pa uporabljamo 12-bitno resolucijo, pa se napetost 3.3 volte pretvori v vrednost 4095. Z večjo resolucijo pridobimo večjo natančnost. V našem primeru bomo uporabljali 6-bitno resolucijo, razloge za to pa bomo spoznali kasneje.

Najprej moramo nastaviti delovanje naprave GPIO (izsek kode 4.6). Napravi GPIO C sporočimo, da bomo na priključku številka 2 brali analogno vrednost.

V izseku kode 4.7 nastavimo delovanje naprave *ADC3*. Priključek številka 2 na napravi GPIO C je povezan na vse tri analogno-digitalne pretvornike na kanal številka 12. S parametrom `ADC_ExternalTrigConv_T3_TRGO` analogno-digitalnemu pretvorniku povemo, naj ob dogodku `TIM3_TRGO` začne s pretvorbo. S klicem ukaza `ADC_DMARequestAfterLastTransferCmd` lahko omogočimo prenos pretvorjene vrednosti preko krmilnika za neposreden dostop do pomnilnika na določeno mesto v delovni pomnilnik.

Izsek kode 4.6: Nastavitev naprave GPIO C.

```
1  RCC_AHB1PeriphClockCmd( RCC_AHB1Periph_GPIOC, ENABLE );
2
3  GPIO_InitTypeDef GPIO_InitStructure2;
4  GPIO_InitStructure2.GPIO_Pin = GPIO_Pin_2;
5  GPIO_InitStructure2.GPIO_Mode = GPIO_Mode_AN;
6  GPIO_InitStructure2.GPIO_Speed = GPIO_Speed_100MHz;
7  GPIO_InitStructure2.GPIO_OType = GPIO_OType_PP;
8  GPIO_InitStructure2.GPIO_PuPd = GPIO_PuPd_NOPULL;
9
10 GPIO_Init( GPIOC, &GPIO_InitStructure2 );
```

### 4.2.3 Krmilnik za neposreden dostop do pomnilnika

V funkciji `initDMA` nastavimo delovanje krmilnika za neposreden dostop do pomnilnika (izsek kode 4.8). Naš sistem vsebuje dva krmilnika za neposre-

Izsek kode 4.7: Nastavitev analogno-digitalnega pretvornika.

```
RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC3, ENABLE );

ADC_CommonInitTypeDef ADC_1;
ADC_1.ADC_Mode = ADC_Mode_Independent;
ADC_1.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;
ADC_1.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
ADC_1.ADC_Prescaler = ADC_Prescaler_Div8;
ADC_CommonInit( &ADC_1 );

ADC_InitTypeDef ADC_2;
ADC_2.ADC_Resolution = ADC_Resolution_6b;
ADC_2.ADC_ScanConvMode = DISABLE;
ADC_2.ADC_ContinuousConvMode = DISABLE;
ADC_2.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T3_TRGO;
ADC_2.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_Rising;
ADC_2.ADC_DataAlign = ADC_DataAlign_Right;
ADC_2.ADC_NbrOfConversion = 1;
ADC_Init( ADC3, &ADC_2 );

ADC_RegularChannelConfig(ADC3, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);

ADC_DMARequestAfterLastTransferCmd( ADC3, ENABLE );

ADC_DMACmd( ADC3, ENABLE );
ADC_Cmd( ADC3, ENABLE );
```

den dostop do pomnilnika, uporabili pa bomo krmilnik *DMA2*. Z ukazom `RCC_AHB1PeriphClockCmd` krmilniku najprej vključimo urin signal, nato pa znotraj strukture tipa `DMA_InitTypeDef` določimo ustrezne vrednosti vseh parametrov.

Rezervirana beseda `__IO`, ki jo najdemo pred definicijo globalne spremenljivke `uint8_t ADCValueEKG`, prevajalniku sporoči, naj je ne poskuša optimizirati. Ker vrednosti te spremenljivke nikjer v programski kodi ne nastavljamo, se lahko zgodi, da prevajalnik sklepa, da te spremenljivke v resnici ne potrebujemo in jo odstrani iz kode.

Vsak izmed obeh krmilnikov za neposreden dostop do pomnilnika ima nadzor nad 8 tokovi, vsak tok pa lahko uporablja do 8 kanalov. Naprava *ADC3* je povezana na kanal številka 2 v toku številka 0, zato tudi uporabljen krmilnik nastavimo v skladu s tem.

Krmilnik za neposreden dostop do pomnilnika lahko uporabljamo za prenos podatkov v različnih smereh. Podatke lahko prenašamo iz delovnega pomnilnika v periferno napravo, iz periferne naprave v delovni pomnilnik ali pa iz enega dela delovnega pomnilnika v drugega. V našem primeru bomo podatke prenašali iz periferne naprave v delovni pomnilnik. Naslov periferne naprave je definiran pod imenom `ADC3_DR_ADDRESS`, njegova vrednost pa znaša `0x4001224C`.

## 4.3 Zaznavanje QRS kompleksa

Funkcija `vTaskHeartBeat` implementira opravilo za zaznavanje QRS kompleksa (izsek kode 4.9). Algoritem za zaznavanje QRS kompleksa je dokaj nezapleten. Temelji na odvodu funkcije elektrokardiograma, kjer merimo razliko med trenutno in prejšnjo vrednostjo. V kolikor je ta razlika manjša od neke določene meje, smo zaznali potencialni QRS kompleks. Da bi preprečili morebitno zaznavanje večih QRS kompleksov hkrati, mora biti število milisekund od prejšnjega zaznanega QRS kompleksa večje od 100.

Ko zaznamo QRS kompleks, izračunamo frekvenco srčnega utripa. To

Izsek kode 4.8: Inicializacija krmilnika za neposreden dostop do pomnilnika.

```
__IO uint8_t ADCValueEKG;

void initDMA() {
   RCC_AHB1PeriphClockCmd( RCC_AHB1Periph_DMA2, ENABLE );

   DMA_InitTypeDef DMA_InitStructure;

   DMA_InitStructure.DMA_Channel = DMA_Channel_2;
   DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) ADC3_DR_ADDRESS;
   DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t) &ADCValueEKG;
   DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
   DMA_InitStructure.DMA_BufferSize = 1;
   DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
   DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
   DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
   DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
   DMA_InitStructure.DMA_Priority = DMA_Priority_High;
   DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
   DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
   DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
   DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;

   DMA_Init( DMA2_Stream0, &DMA_InitStructure );

   DMA_Cmd( DMA2_Stream0, ENABLE );
}
```

storimo tako, da vrednost 60000 delimo s številom pretečenih milisekund. Dobljeno vrednost nato pretvorimo iz številske oblike v tabelo posameznih številk. Kazalec na začetek te tabele nato vstavimo v vrsto. V kolikor je vrsta polna, se bo opravilo postavilo v stanje *Blocked*, dokler v vrsti ne bo dovolj prostora.

Po končanem vstavljanju kazalca na tabelo v vrsto se opravilo s klicem funkcije `vTaskDelay` postavi v stanje *Blocked*.

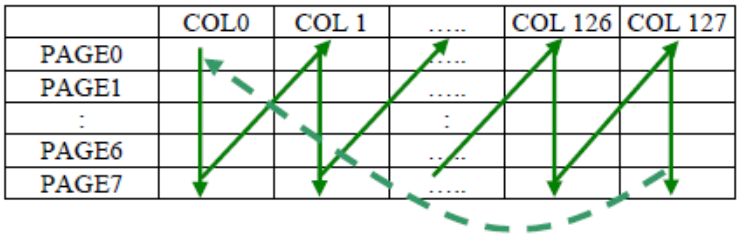
## 4.4 Izrisovanje na zaslon

Funkcija `vTaskZaslonMain` implementira opravilo, ki skrbi za posodabljanje slike na zaslonu. Komunikacija z zaslonom poteka preko protokola SPI. Zaslon je po višini razdeljen na 8 strani, po širini pa na 128 stolpcev. Vsaka stran vsebuje en stolpec višine 8 pikslov. Preden lahko začnemo z risanjem na zaslon, moramo čipu, ki nadzoruje zaslon, sporočiti začetek in konec območja, v katerem bomo risali. Da določimo to območje, moramo poslati šest ukazov. Območje risanja nastavimo z ukazoma `SSD1306_COLUMNADDR` in `SSD1306_PAGEADDR`, vsakemu ukazu pa morata slediti dva podatka o začetku in koncu območja. Prvi ukaz uporabljamo za določitev vrstic, drugi ukaz pa za izbiro strani, v katere bomo risali. Ko nastavimo območje risanja, lahko začnemo s pošiljanjem slike, ki se bo izrisala na izbranem območju na zaslonu. Vsak poslani podatek je dolg 8 bitov, z njim pa hkrati posodobimo eno stran, kar pomeni, da vsak bit odraža stanje enega piksla v posamezni strani. V kolikor bi želeli zapolniti cel zaslon, bi morali na čip poslati 1024 8-bitnih podatkov, kar znaša natanko en kibibajt. Vsakič, ko na čip pošljemo podatek, ki predstavlja del slike, se notranji kazalec na čipu premakne v ustrezno smer, kot je to prikazano na sliki 4.1. Ta sicer prikazuje pomikanje kazalca na območju celotnega zaslona. V primeru, da območje risanja ni enako celotnemu zaslonu, se kazalec premika znotraj meja izbranega območja.

Pred vstopom v neskončno zanko na zaslon izrišemo osnovne dele (izsek

Izsek kode 4.9: Zaznavanje QRS kompleksa.

```
uint8_t ekg, ekgPrev = 0, heartBeat[3], *pointer;

while(1) {
    ekg = ADCValueEKG;

    // Nastavimo mejo za zaznavo utripa
    // Hkrati pa mora od prejšnega utripa srca miniti več kot 100 ms
    // Tako lahko preprečimo zaznave neveljavnih utripov zaradi šuma
    if( ekg - ekgPrev < -6 && heartBeatMS > 100 ) {
        uint8_t i;

        // Število pretečenih milisekund prepišemo v lokalno spremenljivko
        // Hkrati še ponastavimo heartBeatMS na 0
        uint16_t currentHeartBeatMS = heartBeatMS;
        heartBeatMS = 0;

        // Milisekunde pretvorimo v frekvenco srčnega utripa
        currentHeartBeatMS = 60000 / currentHeartBeatMS;

        // Frekvenco nato pretvorimo iz vrednosti v posamezne številke
        // Sicer v nasprotnem vrstnem redu
        // 126 tako postane [ 6, 2, 1 ]
        for( i = 0; i < 3; i++ )  {
            heartBeat[ i ] = currentHeartBeatMS % 10;
            currentHeartBeatMS /= 10;
        }

        // Ko izračunamo podatek o frekvenci srčnega utripa
        // ga preko vrste pošljemo opravilu, ki izrisuje na zaslon
        pointer = heartBeat;
        xQueueSendToBack( queueHeartBeat, &pointer, portMAX_DELAY );
    }

    ekgPrev = ekg;

    vTaskDelay( EKG_REFRESH_RATE_MS * 2 / portTICK_RATE_MS );
}
```




Slika 4.1: Samodejno pomikanje kazalca ob pisanju v pomnilnik zaslona. Povzeto po [20].

kode 4.10). Najprej zapolnimo zgornji del zaslona v širini 128 pikslov in višini 16 pikslov. Na skrajnem levem robu izrišemo ikono srca, desno od nje pa bomo kasneje izpisali frekvenco srčnega utripa. Ob vsakem srčnem utripu bomo uporabnika o tem obvestili z začasno spremembo ikone srca. Na koncu še nastavimo vertikalno območje prikazovanja elektrokardiograma.

Funkcija `send_data` skrbi za pošiljanje podatkov na zaslon preko protokola SPI. Prvi argument te funkcije določa, ali naj čip prejete podatke obravnava kot ukaz ali kot podatke, ki posledično določajo sliko na zaslonu. Drugi argument funkcije pa vsebuje podatek, ki se dejansko pošlje na zaslon.

Izris podatka, ki ga preko analogno-digitalnega pretvornika pridobimo iz modula EKG, lahko vidimo na izseku kode 4.11. Celoten izsek kode, razen deklaracije spremenljivk `pixel_current`, `pixel_prev`, `pixel` in `pozicija`, se nahaja v neskončni zanki znotraj funkcije, ki implementira opravilo za komunikacijo z zaslonom.

Ko zaključimo z izrisovanjem trenutne vrednosti na zaslonu, preverimo vrsto za nove podatke. V kolikor v vrsti najdemo nov podatek, izpišemo novo vrednost frekvence srčnega utripa v levi zgornji kot. Podatek v vrsti je kazalec na tabelo, v kateri najdemo tri števila tipa `uint8_t`, ta tri števila pa potem izpišemo na zaslon. Programsko kodo, ki opravlja to nalogo, najdemo na izseku kode 4.12.

Po končanem izrisovanju opravilo blokiramo za `EKG_REFRESH_RATE_MS` milisekund s klicem funkcije `vTaskDelay`. Argument te funkcije je število

Izsek kode 4.10: Začetno izrisovanje na zaslon.

```
// Zapolnimo zgornji del
// 2 strani širine 128
send_data( SPI_COMMAND, SSD1306_COLUMNADDR );
send_data( SPI_COMMAND, 0 );
send_data( SPI_COMMAND, 127 );
send_data( SPI_COMMAND, SSD1306_PAGEADDR );
send_data( SPI_COMMAND, 0 );
send_data( SPI_COMMAND, 1 );

for( i = 0; i < 128*2; i++ ) {
    send_data( SPI_DATA, 0xFF );
}

// Narišemo inverz srca (črno srce)
// Lokacija: (0,0)
// Širina: 16 px
// Višina: 2 strani (16 px)
send_data( SPI_COMMAND, SSD1306_COLUMNADDR );
send_data( SPI_COMMAND, 0 );
send_data( SPI_COMMAND, 15 );
send_data( SPI_COMMAND, SSD1306_PAGEADDR );
send_data( SPI_COMMAND, 0 );
send_data( SPI_COMMAND, 1 );

for( i = 0; i < 16*2; i++ ) {
    send_data( SPI_DATA, ~srce[i] );
}

// Nastavimo območje izrisovanja na sredino
send_data( SPI_COMMAND, SSD1306_PAGEADDR );
send_data( SPI_COMMAND, 3 );
send_data( SPI_COMMAND, 6 );
```


Slika 4.2: Zaslonska slika zaslona OLED.

dogodkov *systick*, za kolikor se opravilo postavi v stanje *Blocked*.

Končni rezultat risanja na zaslon lahko vidimo na sliki 4.2.

Izsek kode 4.11: Izrisovanje elektrokardiograma.

```
uint32_t pixel_current, pixel_prev, pixel;
uint8_t pozicija = 0;
// Na območju na zaslonu imamo v vsakem stolpcu 32 pikslov po višini
// Spremenljivka ADCValueEKG vsebuje 6-bitno vrednost (2^6 = 64)
// To popravimo tako, da spremenljivko zamaknemo za en bit v desno
// Najbolj desni bit bo izpadel, ostalo nam bo še 5 bitov (2^5 = 32)
uint8_t ekg = ADCValueEKG >> 1;

// Spremenljivka pixel_current je 32-bitna spremenljivka
// Postavimo tisti bit, ki ustreza vrednosti iz EKG
// Hkrati pa še zrcalimo bitno vrednost spremenljivke
pixel_current = 1 << (31 - ekg);

// Risanje črt med piksli, ko je skok med piksli prevelik
// Za to potrebujemo prejšno vrednost
pixel = pixel_current;

if( pixel_current > pixel_prev ) {
    pixel = pixel_current - pixel_prev;
} else if( pixel_current < pixel_prev ) {
    pixel = pixel_prev - pixel_current;
}
pixel_prev = pixel_current;

// Pobrišemo nekaj stolpcev desno od trenutne pozicije
send_data( SPI_COMMAND, SSD1306_COLUMNADDR );
send_data( SPI_COMMAND, (pozicija + 8) % 128 );
send_data( SPI_COMMAND, (pozicija + 8) % 128 );
// V vertikali imamo 4 strani, zato moramo 0x00 poslati 4-krat
for( i = 0; i < 4; i++ ) {
    send_data( SPI_DATA, 0x00 );
}

send_data( SPI_COMMAND, SSD1306_COLUMNADDR );
send_data( SPI_COMMAND, pozicija );
send_data( SPI_COMMAND, pozicija );
// Ker na zaslon izrisujemo po 8 bitov hkrati (po vertikali)
// moramo izris razdeliti na 32/8 = 4 dele
for( i = 0; i < 4; i++ ) {
    // Izrišemo desnih 8 bitov spremenljivke pixel
    send_data( SPI_DATA, (uint8_t) pixel );
    // Nato spremenljivko pixel pomaknemo za 8 bitov v desno
    // da bomo lahko izrisali naslednjih 8 bitov
    pixel = pixel >> 8;
}
// Povečamo položaj v X osi in po potrebi ponastavimo na 0
if( pozicija++ == 127 ) {
    pozicija = 0;
}
```

Izsek kode 4.12: Izpisovanje frekvence srčnega utripa.

```
uint8_t *heartBeat;
// Če je prišlo do srčnega utripa,
// se podatek o njegovi frekvenci nahaja v vrsti
portBASE_TYPE status = xQueueReceive( queueHeartBeat, &heartBeat, 0 );
if( status == pdTRUE ) {
    send_data( SPI_COMMAND, SSD1306_COLUMNADDR );
    send_data( SPI_COMMAND, 16 );
    send_data( SPI_COMMAND, 16*4 - 1 );
    send_data( SPI_COMMAND, SSD1306_PAGEADDR );
    send_data( SPI_COMMAND, 0 );
    send_data( SPI_COMMAND, 1 );

    // Pobrišemo prejšno vrednost
    for( i = 0; i < 16*3*2; i++ ) {
        send_data( SPI_DATA, 0xFF );
    }

    // Če je prva številka 0, jo preskočimo
    // Tako bomo namesto 012 izpisali 12
    // Številke se v tabeli nahajajo ravno v nasprotnem vrstnem redu
    // Prva številka se tako nahaja na zadnjem mestu v tabeli
    if( heartBeat[2] == 0 ) {
        i = 1;
    } else {
        i = 2;
    }

    for( i; i >= 0; i-- ) {
        // Vsaka številka je široka 12 pikslov
        // Vendar pustimo 2 piksla na levi in na desni strani
        send_data( SPI_DATA, 0xFF );
        send_data( SPI_DATA, 0xFF );

        for( j = 0; j < 12; j++ ) {
          send_data( SPI_DATA, ~stevilke[ heartBeat[i] ][j] );
        }

        send_data( SPI_DATA, 0xFF );
        send_data( SPI_DATA, 0xFF );
    }
}

// Opravilo blokiramo za določeno število milisekund
// Hladna sinhronizacija z ADC
vTaskDelay( EKG_REFRESH_RATE_MS / portTICK_RATE_MS );
```

# Poglavje 5

# Sklepne ugotovitve

V okviru diplomske naloge smo razvili platformo za prikaz in analizo elektrokardiograma, ki temelji na razvojni ploščici STM32F4Discovery. Najprej smo na kratko pregledali področje elektrokardiografije in primere njene uporabe. Opisali smo uporabljeno strojno in programsko opremo, na koncu pa smo še razčlenili razvito programsko implementacijo. Kljub preprosti implementaciji pa je naša platforma zelo fleksibilna, kar smo zagotovili s pametnim načrtovanjem in uporabo operacijskega sistema FreeRTOS.

Problem elektrokardiografije ostaja v dejstvu, da je za uspešno prepoznavo morebitnih obolenj srčne mišice potrebno pridobiti ustrezno izobrazbo. Za razliko od merjenja krvnega tlaka mora analizo elektrokardiograma še vedno opraviti ustrezno usposobljena oseba.

Uporabljeni modul EKG ni certificirana medicinska naprava, zato je natančnost dobljenega elektrokardiograma vprašljiva. V samem elektrokardiogramu se lahko pojavljajo motnje in šum, kar onemogoča natančnejšo analizo dejavnosti srčne mišice. To težavo bi lahko rešili z uporabo medicinske EKG naprave, poleg večje natančnosti pa bi s tako napravo pridobili tudi na številu odvodov.

Med razvojem smo imeli v nekaterih zgradbah nekaj težav zaradi slabe ozemljenosti električne napeljave, saj smo v elektrokardiogramu lahko zaznali nihanje funkcije s frekvenco nihanja omrežne napetosti.

Uporabljeni algoritem za zaznavo QRS kompleksa je dokaj preprost. Navkljub preprostosti pa je algoritem na elektrokardiogramu zdravega človeka uspešno zaznaval QRS komplekse. Prostora za izboljšave je še veliko, saj v trenutni izvedbi procesor večino časa ne počne ničesar. Z uporabo posebnih procesorskih ukazov za digitalno procesiranje signalov (angl. *DSP - digital signal processing*) bi lahko implementirali naprednejše algoritme za zaznavo QRS kompleksa, ki so veliko robustnejši na motnje v elektrokardiogramu.

# Literatura


[1] Wikipedia. Electrocardiography — wikipedia, the free encyclopedia, 2015. [Online; accessed 3-August-2015].

[2] Ivan Tomasic and Roman Trobec. Electrocardiographic systems with reduced numbers of leads—synthesis of the 12-lead ecg. *Biomedical Engineering, IEEE Reviews in*, 7:126–142, 2014.

[3] Jan A Kors and Gerard van Herpen. How many electrodes and where? a [ldquo] poldermodel [rdquo] for electrocardiography. *Journal of electrocardiology*, 35(4):7–12, 2002.

[4] ECGpedia. Basics, 2015. [Online; accessed 3-August-2015].

[5] Libardo J Melendez, DT Jones, and JR Salcedo. Usefulness of three additional electrocardiographic chest leads (v7, v8, and v9) in the diagnosis of acute myocardial infarction. *Canadian Medical Association Journal*, 119(7):745, 1978.

[6] Robert J Zalenski, David Cooke, Robert Rydman, Edward P Sloan, and Daniel G Murphy. Assessing the diagnostic value of an ecg containing leads v 4r, v 8, and v 9: the 15-lead ecg. *Annals of emergency medicine*, 22(5):786–793, 1993.

[7] Heikki V Huikuri, Timo H Mäkikallio, Chung-Kang Peng, Ary L Goldberger, Ulrik Hintze, Mogens Møller, et al. Fractal correlation properties of rr interval dynamics and mortality in patients with depressed left

ventricular function after an acute myocardial infarction. *Circulation*, 101(1):47–53, 2000.

[8] Wikipedia. Qrs complex — wikipedia, the free encyclopedia, 2015. [Online; accessed 3-August-2015].

[9] Katja Ažman Juvan and Petra Zupet. The athlete's electrocardiogram. *Slovenian Medical Journal*, 79(9), 2010.

[10] Fumihiko Yasuma and Jun-ichiro Hayano. Respiratory sinus arrhythmia*: Why does the heartbeat synchronize with respiratory rhythm? *Chest*, 125(2):683–690, 2004.

[11] RR ECG. Detection of obstructive sleep apnea in pediatric subjects using surface lead electrocardiogram features. *Sleep*, 27(4):784, 2004.

[12] Yogendra Narain Singh and Phalguni Gupta. Biometrics method for human identification using electrocardiogram. In *Advances in Biometrics*, pages 1270–1279. Springer, 2009.

[13] Dušan Kodek. *Arhitektura in organizacija računalniških sistemov*. Bitim, 2008.

[14] STMicroelectronics. Stm32f4discovery discovery kit with stm32f407vg mcu, 2015. [Online; accessed 7-August-2015].

[15] ARM. Cortex-m4 processor, 2015. [Online; accessed 7-August-2015].

[16] Sparkfun. Sparkfun single lead heart rate monitor - ad8232, 2015. [Online; accessed 14-August-2015].

[17] Adafruit. Monochrome 1.3 128x64 oled graphic display, 2015. [Online; accessed 7-August-2015].

[18] IAR Systems. Iar embedded workbench, 2015. [Online; accessed 14-September-2015].

[19] FreeRTOS. Mutex semaphores with priority inheritance for priority inversion avoidance in mutual exclusion using in freertos real time embedded software applications, 2015. [Online; accessed 26-August-2015].

[20] SOLOMON SYSTECH. Ssd1306, advance information, 128 x 64 dot matrix, oled/pled segment/common driver with controller, 2015. [Online; accessed 5-September-2015].